J Grid Computing (2016) 14:477–493
DOI 10.1007/s10723-015-9351-x

Modelling Fine-Grained Access Control Policies in Grids

Benjamin Aziz

Received: 29 June 2015 / Accepted: 15 September 2015 / Published online: 29 September 2015

Abstract This paper presents an abstract specification of an enforcement mechanism of usage control for Grids, and verifies formally that such mechanism enforces UCON policies. Our technique is based on KAOS, a goal-oriented requirements engineering methodology with a formal LTL-based language and semantics. KAOS is used in a bottom-up form. We abstract the specification of the enforcement mechanism from current implementations of usage control for Grids. The result of this process is agent and operation models that describe the main components and operations of the enforcement mechanism. KAOS is used in top-down form by applying goal-refinement in order to refine UCON policies. The result of this process is a goal-refinement tree, which shows how a goal (policy) can be decomposed into sub-goals. Verification that a policy can be enforced is then equivalent to prove that a goal can be implemented by the enforcement mechanism represented by the agent and operation models.

Keywords Access control · Grid authorisation · Usage control

B. Aziz (✉)
School of Computing, University of Portsmouth,
Portsmouth PO1 3HE UK
e-mail: benjamin.aziz@port.ac.uk

1 Introduction

Grids [1] and Data Grids [2] are technologies providing access to large-scale distributed computing and storage capabilities that span multiple administrative domains and are anchored on a variety of operating systems and technologies. This heterogeneous distribution of resources requires scalable, flexible, and fine-grained access control to protect both individual and shared resources. At the same time, any access control mechanism must be driven by well-structured and well-formulated policy goals capturing the requirements of the resources, their providers and their users.

This paper studies the formulation of the requirements of fine-grained access control enforcement in Grid systems as a case-study in how security policy requirements can be formalised in large-scale distributed systems. Fine-grained access control techniques, or also known as *usage control*, extend traditional access control mechanisms by controlling data and resources usage as well as their access at the entry point [3, 4]. The application of usage control techniques to Grid systems has been demonstrated in the past in works such as [5, 6]. Here we adopt the reference usage control model as proposed by Park and Sandhu in [3], which is also known as the UCON$_{abc}$ model.

Our approach is to define a usage-based Grid authorisation architecture, which uses the functional components of the current Grid systems as outlined by the Open Grid Forum (OGF)[1] group on Grid authorisation (part of the Security Area). We focus on two important aspects in the design of policy-based management systems: the refinement of policies expressed as requirements on the system, and the specification of enforcement mechanisms derived from the above requirements. In formalising the requirements of an enforcement mechanism for Grid UCON$_{abc}$ policies, we adopt the KAOS requirements-engineering methodology [7].

Our approach is to use the KAOS methodology to show how one can abstractly specify a UCON$_{abc}$ policy as a goal or a requirement on the system and its resources. The specification is expressed in the temporal linear logic-based language provided by KAOS. We then derive the KAOS agent and operation models for each of the requirements (polices) specified. This way, we formally show that the enforcement mechanism is sound and complete, and capable of enforcing all the policies pertaining to the UCON$_{abc}$ family of core models. For reasons of conciseness, we limit ourselves in this paper to the refinement of a couple of UCON$_{abc}$ models, namely the PreA0 and OnA3 models [8]. Nonetheless, the approach is general to be applied to the rest of the UCON$_{abc}$ family of models.

The rest of the paper is structured as follows. In Section 2, we give some background on the UCON model providing an overview of the various elements that constitute the model. We then review the existing literature related to UCON implementations for Grid systems in Section 3, and we define a reference architecture for Grid usage control, which is based on OGF's OGSA architecture. In Section 4, we give an overview of the KAOS requirements engineering methodology and its various models. This methodology is then used to formalise an abstract specification of a Grid UCON enforcement mechanism in Section 5, and in Section 6, we use KAOS to derive the enforcement mechanism operations for a couple of UCON policy examples; PreA$_0$ (Section 6.1) and OnA$_3$ (Section 6.2). Finally, we discuss related work in Section 7 and conclude the paper in Section 8 giving directions for future work.

2 The UCON$_{abc}$ Model

The UCON$_{abc}$ usage control model is a framework defined by Park and Sandhu [3, 9] for the specification of fine-grained access and usage control policies. Henceforth, we simply refer to this framework as UCON. In UCON, subjects (i.e. users) and objects (i.e. computational resources) may have *mutable* as well as immutable attributes, thereby facilitating the *continuity* of the decision-making process when enforcing security policies. This means that the decision to allow a user access to a resource is continuously reviewed before and during the user's access to the resource. The decisions themselves are based not only on *authorisations*, but also on *obligations* and *conditions*. As Grid resources must continuously be monitored for efficient use since the applications that utilise them are generally computationally intensive, the UCON model becomes an attractive security policy solution that can offer fine-grained monitoring and control of such resources.

Elements of the UCON model include:

- *Subjects, Objects* and *Rights*: The subject is the entity that exercises rights, i.e. that executes access operations on objects. An object, instead, is an entity that is accessed by subjects through access operations. Rights are the privileges that subjects can exercise on objects. UCON determines the existence of a right dynamically, whenever a subject attempts to access and exercise a right on some object.
- *Attributes*: Both subjects and objects have attributes. These attributes can be *mutable*, i.e. they can change over time and be updated, or *immutable*, i.e. they are constant and cannot change over time. An example of a mutable attribute is the number of times that a subject accesses an object, whereas an immutable is a subject's or an object's identity.
- *Predicates*: Predicates are logical statements about the subjects' and objects' attributes and the requested right. Predicates can be either *authorisation*, *obligation* or *condition* predicates or any combination of these. Authorisation predicates express set rules that determine whether to grant the requested right or not, and could exploit both attributes of subjects and objects.

[1] Web address: http://www.ogf.org

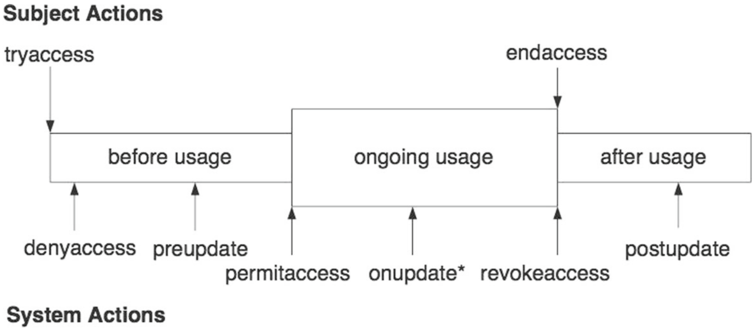


Fig. 1 Access and usage control stages in the UCON model [8]

The evaluation of the predicates can be performed before or during the execution of an action. Obligation predicates are UCON decision factors that are used to verify whether the subject has satisfied some mandatory requirements before performing an action or whether the subject continuously satisfies these requirements while performing the action. Finally, condition predicates are environmental or system-oriented decision factors, i.e. dynamic factors that do not depend on subjects or objects. Condition predicates are evaluated at runtime when the subject attempts to perform the access, and they can be evaluated before or during an action.

The complete UCON$_{abc}$ family of models encompasses several factors. These factors include the presence of Authorisations (A), oBligations (B) and Conditions (C), pre- and on-going decisions, as well as the mutability of attributes (immutable (0), mutable preUpdate (1), mutable onUpdate (2) and mutable postUpdate (3)). The various UCON models differ in the presence of attribute updates and in the sequentiality of the operations taking place. Therefore, an enforcement mechanism for UCON policies must be able to enforce not only the single operations, but the sequence in which these operations are invoked. The different actions that subjects and the system can perform in the UCON model relate to the different phases of an object's usage. Given that the triple (s, o, r) represents the subject s requesting the right r for accessing the object o, we consider the following set of actions [8]:

(i) $tryaccess(s, o, r)$: performed by subject s when performing a new access request,

(ii) $permitaccess(s, o, r)$: performed by the system when granting the access,

(iii) $denyaccess(s, o, r)$: performed by the system when rejecting the access,

(iv) $revokeaccess(s, o, r)$: performed by the system when revoking an ongoing access,

(v) $endaccess(s, o, r)$: performed by s when ending an access,

(vi) $update(s, o, r)$: performed by the system to update a subject's or an object's attributes, anytime before, during or after the access and usage of the object.

The various stages of the UCON access and usage control are shown in Fig. 1.

All the UCON authorisation policies are defined for positive permissions: if there is no policy to enable the permission according to the attribute values, then the access is denied by default. This is sometimes called the closed system assumption, whereby no policy is specified to deny an access in a system. The same holds for obligation and condition core models.

3 Usage Control for Grids

In this section, we first review some reference implementations of UCON for Grids. Then, using notions extracted from the implementations and from the OGF's Open Grids Services Architecture (OGSA) authorisation working group,[2] we introduce the target Grid security architecture for our UCON-policy enforcement mechanism.

[2] Web address: https://redmine.ogf.org/projects/ogsa-authz-wg/

There exists a number of implementations of the UCON model tailored for Grid systems, e.g.[5, 6, 10, 11]. In [5, 10], the authors provide a model of usage control for computational Grids, following Park and Sandhu's UCON$_{abc}$ model [3]. One of the contributions of their work is the use of a special process algebra, called the *POlicy Language based on Process Algebra* (POLPA). POLPA is used as a policy specification language, which is especially suitable to model usage control policies related to the UCON$_{abc}$ model. The process algebra is also capable of expressing the order in which the security-relevant actions are to be performed.

The POLPA-based prototype implements an architecture where the main components are a Policy Enforcement Point (PEP) and a Policy Decision Point (PDP), such as is the architecture in most of the common distributed authorisation systems [12]. The PEP receives a request from an external user and generates a $tryaccess(s, o, r)$ action that it sends to the PDP. Similarly, when the subject ends their access to the object, the PEP generates an $endaccess(s, o, r)$ action to the PDP. The PDP gets the security policy from a repository, exploits its representation and determines whether the access should or should not be allowed, returning to the PEP a $permitaccess(s, o, r)$ or a $denyaccess(s, o, r)$ action. The PDP continuously evaluates a set of given authorisations, conditions and obligations while an access is in progress, and it could invoke the PEP to terminate the access through the $revokeaccess(s, o, r)$ action, if any of the predicates associated with it becomes false.

In [6], Zhang et al. propose a UCON prototype implementation for Grids and collaborative applications, by following a layered approach with policy enforcement and implementation models, called the *Policy-Enforcement-Implementation* (PEI) framework. The security architecture leverages a centralised attribute repository in each virtual organisation and a usage monitor in each resource provider for attribute management. The policies are specified with the eXtensible Access Control Markup Language (XACML) [12], which, as recognised by the same authors, suffers from the impossibility to exactly encode an abstract UCON$_{abc}$ policy, this is despite the fact that in recent years, some works (e.g. [13, 14]) have attempted to use XACML to encode usage control, to varying degrees of success. Within the architecture, both PDP and PEP are located on the resource provider side. For an access, the PDP collects the subject and object attributes, the system attributes, and makes the usage control decision, which is enforced by the PEP. The immutable subject attributes are pushed to the PDP by the requesting subject.

Related to the above POLPA language and policy enforcement architecture, [15], have proposes a couple of methods for testing any implementation of the architecture based on two strategies; a fault-based to uncover vulnerabilities and problems that may occur during a PDP implementation, and the other is based on conditions coverage with a methodology for simulating the continuous control of the PDP during the runtime execution. In [11], an architecture and implementation were proposed for enforcing UCON policies in a Grid system, based on the Globus Grid middleware[3] [16]. The architecture extended that for POLPA, and the language used was also a simplified version of the POLPA policy language.

In [17], the OGSA authorisation working group proposed some functional components for a Grid service provider authorisation service middleware. In their work, great attention was put on *credentials*, defined as attribute assertions digitally signed by the issuer that can be cryptographically validated. Credentials can be issued by Credential Issuing Services (CISs) and validated by Credential Validation Services (CVSs), which return the valid attributes of the subject. Other functional components comprise the usual PDP and PEP components, as well as Context Handlers (CHs) responsible for handling the communications between PEPs, CVSs and PDPs. The interactions between these functional components can be constructed in four different ways, according to whether the credentials and the authorisation decisions are pulled or pushed.

A usage-based Grid authorisation architecture does not require changing the way the functional components interact with one another. As the reader can see in Fig. 2, an access requestor (a Grid User) pushes his/her credentials to a PEP. Then, after the CH obtains valid attributes from the CVS, a PDP is interrogated for an authorisation decision, which in the end is returned to the PEP. From a UCON point of view, valid attributes released by a CVS are examples of *immutable* (persistent) attributes, and would

[3] Web address: https://www.globus.org

include information such as the name or identity of the user. A complex UCON PDP should be able to evaluate policies where the predicates are statements about the subjects' and objects' attributes. Three subcomponents, namely the *Reference Monitor*, the *Predicate Validator* and the *Attribute Manager* make up the UCON PDP. They are explained with details in Section 5.

External components are needed to supply the UCON PDP with the needed information: *(i)* a Virtual Organisation (VO) *UCON policy repository* provides the PDP with the UCON policies to be evaluated, *(ii)* a *meta-data repository* provides the PDP with the optional immutable object attributes, *(iii)* a *VO attributes repository* stores the mutable attributes of the subjects, and finally *(iv)* a *Resource Provider (RP) attributes repository* stores the mutable attributes of the objects.

For an access, the PDP collects the immutable subject and object attributes, as well as search for the UCON policies to be enforced. The policy is selected using the access requestor ID (the UCON subject), and the UCON object requested. Mutable subject and object attributes are pulled by the PDP from the VO's centralized attribute repository, and from the local RP's usage monitor. The updates of mutable subjects' and objects' attributes are performed by the Attribute Manager. Since we only deal with the UCON$_a$ family of core models, the usage-based Grid authorisation architecture does not take into consideration components for *obligations* and *conditions*-based decisions.

4 KAOS or Keep All Objects Satisfied

Knowledge Acquisition in autOmated Specification (KAOS) [18], or sometimes referred to as "Keep All Objects Satisfied", is a generic methodology based on capturing, structuring and precise formulation of system goals [7, 18]. A goal is a prescriptive description of system properties, formulated in non-operational terms. A system includes not only the software to be developed but also its environment. Goals are refined and operationalised in a top-down manner as the system is designed, or with a bottom-up approach while re-engineering existing systems. The approach also supports adverse environments, composed of possibly malicious external agents trying to undermine the system goal rather than to collaborate in the goal fulfilment. As a Grid system is typically composed of a large number of nodes interacting in an open and adverse environment, this approach fits our needs well and although we do not deal here with Grid attacks and vulnerabilities, this would be interesting future research.

The KAOS methodology offers a number of models:

- The **goal model** captures and structures the assumed and required properties of a system by formalising a property as a top-level goal, which is then refined to intermediate subgoals and finally to low-level requirements representing goals that can be operationalised. Goals may be organised in AND/OR refinement-abstraction hierarchies.

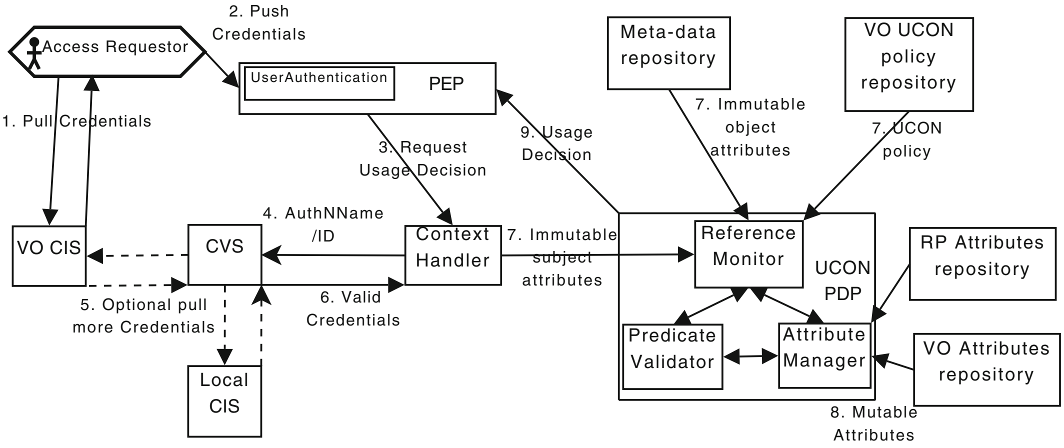


Fig. 2 A usage-based Grid authorisation architecture where credentials are pushed

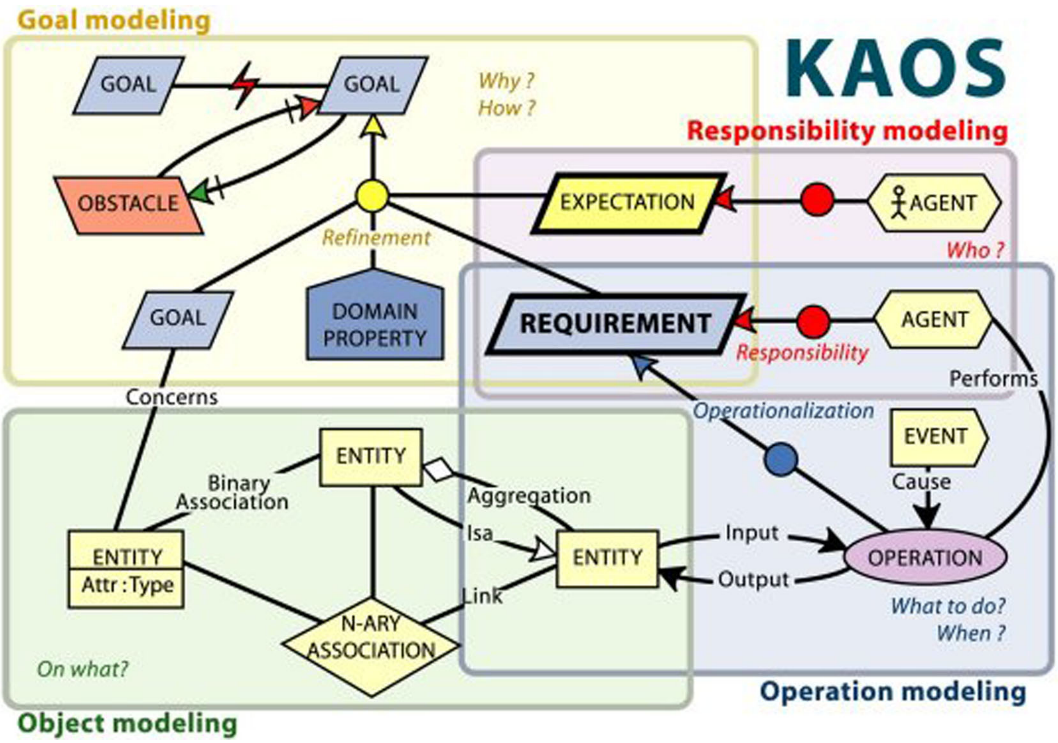


Fig. 3 Overview of the KAOS models and their inter-relationships [20]

AND-refinement links relate a goal to a set of sub-goals possibly conjoined with domain properties or environment assumptions; this means that satisfying all subgoals in the refinement is a sufficient condition in the domain for satisfying the goal. OR-refinement links relate a goal to a set of alternative refinements.

- The **agent model** assigns goals to agents in a realisable way. Agents include software components that exist or are to be developed, external devices, and humans in the environment. Discovering all the responsible agents is one of the criteria for stopping a goal-refinement process.
- The **object model** is used to identify the concepts of the application domain that are relevant to the requirements and to provide static constraints on the operational systems that will satisfy the requirements. The object model consists of objects from the domain, such as any resources and relationships among resources.
- The **operation model** details, at state-transition level, the actions an agent has to perform to realise the goals and requirements it is responsible for.

The KAOS language has a two-layer structure: an outer conceptual modelling layer for declaring concepts (such as goals, objects, agents, etc.) and links between concepts (such as goal refinements, responsibility assignments of goals to agents, etc.), and an inner assertion layer for formally defining concepts. The rigour of the KAOS methodology stems from the fact that any concept defined within its models incorporates formal definitions using Linear Temporal Logic (LTL) [19] formulae. LTL formulae consist of combinations of the usual first-order predicate logic operators ($\wedge \vee \neg \ \rightarrow \leftrightarrow$) along with the following temporal operators expressed on predicates P and Q:

$\square P$, which says that P is always true from now on

$\lozenge P$, which says that P will be true sometime in the future

$\circ P$, which says that P will be true in the next state

$\blacksquare P$, which says that P was always true till now

$\blacklozenge P$, which says that P was true at sometime in the past

$\bullet P$, which says that P was true in the previous state
$P\mathcal{S}Q$, which says that Q has been true since a time when P was true
$P\mathcal{U}Q$, which says that Q will be true until a time when P will be true

We also write a couple of shorthand notations: $(P \Rightarrow Q)$ to mean $\Box(P \rightarrow Q)$ and $(P \Leftrightarrow Q)$ to mean $(P \Rightarrow Q) \wedge (P \Leftarrow Q)$, and we also utilise the bounded forms of all of the above operators, as we shall see in Section 6.2.

Figure 3 shows an overview of the four KAOS models and their inter-relations.

5 An Abstract Specification of a Grid UCON Enforcement Mechanism

Our main focus in this paper is on the sub-family of UCON models known as UCON$_a$, which is concerned with controlling authorisation decisions only (i.e. neither obligations nor condition factors are considered). Henceforth, we use the terms UCON and UCON$_a$ interchangeably. In this section, we define an abstract specification of an enforcement mechanism for UCON$_a$ policies using the LTL-based requirement specification language provided by KAOS. The specification has been partially abstracted from the usage-based authorisation architecture of Section 3, while the operations are inferred from UCON's formal definitions presented in [8] and overviewed in Section 2. Figure 4 shows an illustration of the UCON PDP components as KAOS agents, and the operations those components (agents) can perform. We identify three agents in addition to the **Subject** agent (i.e. the Grid user):

i) **Attribute Manager** (AM), which updates the attributes and returns their values,
ii) **Predicate Validator** (PV), which validates the policy predicates, and
iii) **Reference Monitor** (RM), which is a gateway for all the authorisation decisions generated by the UCON policy enforcement mechanism.

The RM is responsible for performing the *PermitAccess*, *DenyAccess* and *RevokeAccess* operations. These operations are performed in response to a *TryAccess* performed by the subject. The subject also performs an *EndAccess*, which corresponds to an application cleanly signalling its end of use of the Grid resources. The PV can be invoked for the validation of the predicates, viz. performing the *PredicateValidation* operation. The AM can be invoked for the update of the UCON attributes with the *AttributeUpdate* operation.

We now provide a written operational software specification of most of the operations shown in Fig. 4 using the KAOS operation model. We do not specify *TryAccess* nor *EndAccess* since they are subject-specific operations outside the scope of the enforcement mechanism; part of the environment (context) of the mechanism. Each operation defines a state-transition in the application domain, defined through *domain pre-* and *post-conditions*. Operations have input and output fields; for example, *subject*, *object* and *right* are input parameters to the operations *PermitAccess*, *DenyAccess* and *RevokeAccess*.

Operation: PermitAccess
Perf By: Reference Monitor
Domain Pre-Condition:
¬ RM.permitaccess(s, o, r)
Domain Post-Condition:
RM.permitaccess(s, o, r)
: subject, object, right

Operation: DenyAccess
Perf By: Reference Monitor
Domain Pre-Condition:
¬ RM.DenyAccess(s, o, r)
Domain Post-Condition:
RM.DenyAccess(s, o, r)
Input: subject, object, right

Operation: RevokeAccess
Perf By: Reference Monitor
Domain Pre-Condition:
¬ RM.RevokeAccess(s, o, r)
Domain Post-Condition:
RM.RevokeAccess(s, o, r)
Input: subject, object, right

Operation: PredicateValidation
Perf By: Predicate Validator
Domain Pre-Condition:
¬ PV.validate($p_1 \wedge \ldots \wedge p_n$)
Domain Post-Condition:
PV.validate($p_1 \wedge \ldots \wedge p_n$)
Input: Set of Predicates $p_1, \ldots, p_n$
Output: ValidationResponse

Operation: AttributeUpdate
Perf By: Attribute Manager
Domain Pre-Condition:
Domain Post-Condition:
AM.update(s, o, r)
Input: Attribute, Operation, Value
Output: Attribute, Value

In KAOS, an important distinction is made between descriptive domain *pre-* and *post-conditions* and prescriptive *required pre-*, *post-* and *trigger conditions*. The *required pre-condition* captures a permission to perform the operation only if the condition is true. By contrast, the *required post-condition* defines some additional conditions that any application of the operation must establish. The *required trigger condition* captures an obligation to perform the operation if the condition becomes true provided the

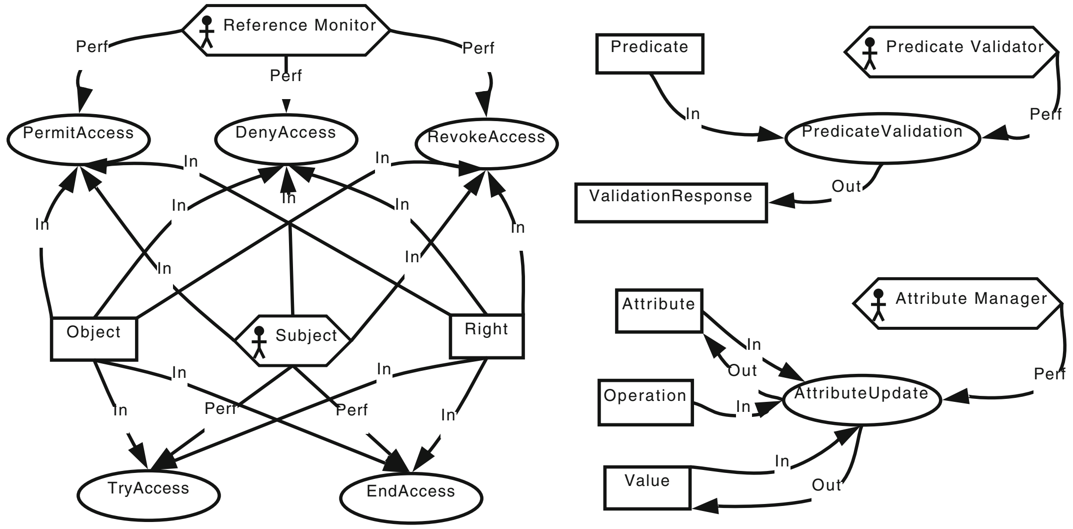


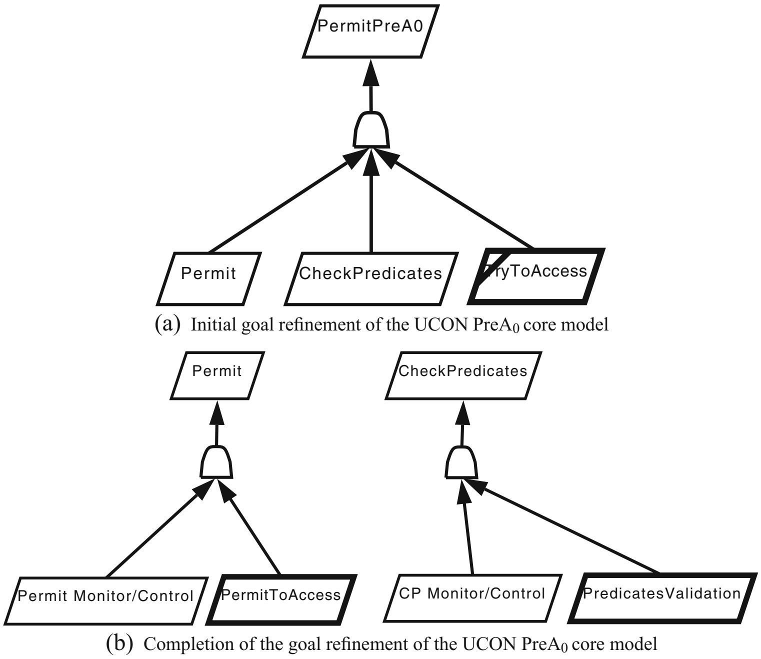
Fig. 4 UCON$_a$ enforcement mechanism expressed in terms of KAOS agents and operations

domain precondition is true. Domain conditions are a property of the operation itself hence they have a static definition, whereas the required conditions relate to the satisfaction of a specific goal/requirement and hence their definition varies according to the goal/requirement the operation is assigned to.

6 Using KAOS as a Formal Specification Proof Language

Policy refinement is concerned with the transformation of a high-level abstract policy specification into a low-level concrete format that can be directly enforced

Fig. 5 Goal model for the UCON PreA$_0$ core model



(a) Initial goal refinement of the UCON PreA$_0$ core model

(b) Completion of the goal refinement of the UCON PreA$_0$ core model

[21]. The policy refinement process includes, generally, the following three steps:

(1) Determining the resources that are needed to satisfy the requirements of a policy
(2) Translating the high-level policies into operational policies that can be enforced
(3) Verifying that the lower level policies actually meet the requirements specified at the higher levels

Here, we follow the goal-based approach to policy refinement introduced by Bandara et al. in [22], which uses the KAOS goal-refinement methodology. KAOS is appropriate for this task since it includes a rigours notation for representing goals and strategies to refine a goal into a set of subgoals, and ultimately implementable requirements. The refined subgoals imply the parent goal and are more detailed.

A goal refinement is correct if it is *complete*, *consistent*, and *minimal*. A set of goals $\{G_1, G_2, \ldots, G_n\}$ correctly refines a parent goal G under some domain assumptions and properties, D, if the following corresponding conditions hold:

$$G_1, \ldots, G_n, D \Rightarrow G \qquad \text{(completeness)}$$

$$G_1, \ldots, G_n, D \not\Rightarrow false \qquad \text{(consistency)}$$

$$\bigwedge_{j \neq i} G_j, D \not\Rightarrow G \text{ for any } i \in [1..n] \qquad \text{(minimality)}$$

More informal explanation of these properties can be found in [18]. Verifications can then be made on goal refinements to ensure that the system meets the goals and that the goal model is well-formed with respect to the above properties.

Our approach is to define how a UCON$_a$ policy can be enforced in terms of the required operations and agents, given its definition as a KAOS goal. More specifically, we can state that our top-level goal is as follows:

$$\forall s : subject, o : object, r : right$$
$$permitaccess(s, o, r) \Rightarrow policyEnforcing(s, o, r)$$

Where *policyEnforcing(s, o, r)* is a predicate that states that the UCON policy in place is being enforced and it may have various forms depending on the particular UCON model adopted. This goal states that if a request from a subject to an object to execute a right is to be permitted, it must be the case that the UCON policy in place to protect the object is being enforced.

We also define *when* to enforce a policy. For example, each policy pertaining to a PreA$_0$ model need to be enforced only before the access is actually granted, while policies pertaining to a PreA$_3$ model need to be enforced not only before the access, but also after the access has ended. Moreover, for the case of OnA-type policies, the policy must also be enforced during the access period. To demonstrate our approach, in this paper we only consider two examples of policies: $PreA_0$ and OnA_3.

6.1 Refinement and Operationalisation of the UCON PreA$_0$ Policy Model

In the PreA$_0$ model, a usage control decision is determined by authorisations before the usage, and there are no attribute updates before, during, or after this usage. Discretionary Access Control (DAC) with Access Control Lists (ACLs) is a classical example of such PreA$_0$-type of policies. An immutable subject attribute could be its identity while an immutable object attribute could be an ACL, *acl*, which consists of pairs (id, r), where id refers to a subject's identity, and r refers to a right with which the subject can access the object. The predicate to be satisfied

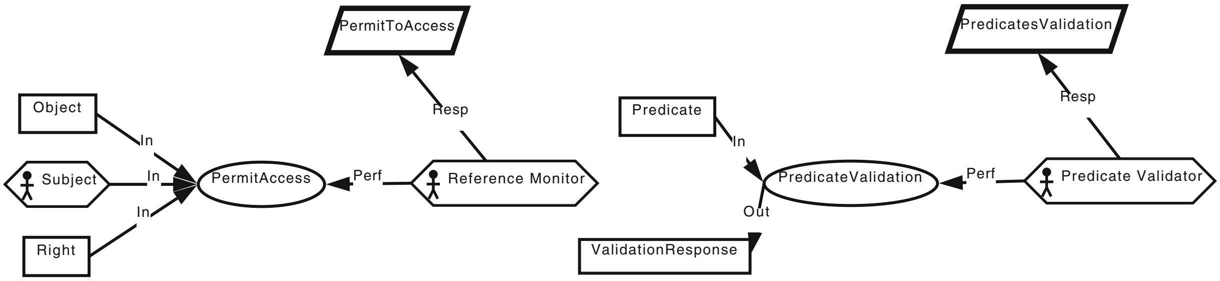


Fig. 6 The operation model for UCON PreA$_0$ showing the agent/responsibility relations

in order for access to be granted can be written as $((s.id, r) \in o.acl)$.

We require the policy (expressing the above predicate) to be enforced in some state prior to when the access was permitted. Therefore we write the top goal as:

Goal [PermitPreA0]
RefinedTo: [Permit], [CheckPredicates], [TryToAccess]
FormalDef: ($\forall$ s:subject, o:object, r:right)
permitaccess(s, o, r) $\Rightarrow \blacklozenge$ policyEnforcing(s, o, r)

Where policyEnforcing(s, o, r) is a predicate stating that the ACL policy is being enforced. We apply a first goal refinement, as shown in Fig. 5a, where the formal sub-goals' definitions follow. We can use tools, such as the FAUST toolkit [23], to demonstrate that the refinement is correct.

Goal [Permit]
Refines: [PermitPreA0]
RefinedTo: [Permit Monitor/Control],
[PermitToAccess]
FormalDef: ($\forall$ s:subject,
o:object,
r:right)
permitaccess(s, o, r) $\Rightarrow \bullet$ ($p_1 \wedge \ldots \wedge p_n$)

Goal [CheckPredicates]
Refines: [PermitPreA0]
RefinedTo: [CP Monitor/Control],
[PredicatesValidation]
FormalDef: ($\forall$ s:subject,
o:object,
r:right)
($p_1 \wedge \ldots \wedge p_n$) $\Rightarrow \bullet$ tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [PermitPreA0]
FormalDef: ($\forall$ s:subject, o:object, r:right)
tryaccess(s, o, r)$\Rightarrow \bullet$ policyEnforcing(s, o, r)

[TryToAccess] is a final goal, in this case, a requirement on the domain of the system (i.e. an assumption of the system). However, both [Permit] and [CheckPredicates] are sub-goals that can be further refined. [Permit] states for the access to be permitted, the predicates controlling that access must be valid in the state before access is granted. This then acts as a milestone for the next sub-goal, [CheckPredicates], which further states that in the previous state to the state when the predicates were valid, a request must have been received. Finally, the domain assumption is that this request implies that the policy was being enforced in the previous state. These three sub-goals together fulfil the parent top goal.

In Fig. 5b, the completion of the goal refinement is shown, and the formal definitions of each of the sub-goals follows in the text. We identify two requirement goals, [PermitToAccess] and [PredicatesValidation], and assign two agents, the *Reference Monitor* and the *Predicate Validator* to respectively take care to each of these. The other two sub-goals become assumptions on the domain; [Permit Monitor/Control] states that permitting access is performed by the reference monitor, and [CP Monitor/Control] states that the validation of the relevant predicates on subject and object attributes is performed by the predicate validation component. [PermitToAccess] then refines [Permit] to include the role of the RM and PV components, and similarly, [PredicatesValidation] also refines [CheckPredicates] to include the role of the PV component.

Goal [Permit Monitor/Control]
Refines: [Permit]
FormalDef: ($\forall$ s:subject, o:object, r:right,
RM:Reference Monitor,
PV:Predicate Validator)
permitaccess(s, o, r) $\Leftrightarrow$ RM.permitaccess(s, o, r)

Goal [PermitToAccess]
Refines: [Permit]
FormalDef: ($\forall$ s:subject, o:object, r:right,
RM:Reference Monitor,
PV:Predicate Validator)
RM.permitaccess(s, o, r)
$\Rightarrow \bullet$ PV.validate($p_1 \wedge \ldots \wedge p_n$)
Resp: Reference Monitor

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: ($\forall$ s:subject, o:object, r:right,
PV:Predicate Validator)
($p_1 \wedge \ldots \wedge p_n$) $\Leftrightarrow$ PV.validate($p_1 \wedge \ldots \wedge p_n$)

Goal [PredicatesValidation]
Refines: [CheckPredicates]
FormalDef: ($\forall$ s:subject, o:object, r:right,
PV:Predicate Validator)
PV.validate($p_1 \wedge \ldots \wedge p_n$)
$\Rightarrow \bullet$ tryaccess(s, o, r)
Resp: Predicate Validator

We are now capable of deriving the KAOS agent and operation models. Figure 6 shows the KAOS operation model, together with the agent/responsibility model. As the reader can see, we identify a couple of operations: `PermitAccess` and `PredicateValidation`, which operationalise the above requirements [PermitToAccess] and [PredicatesValidation], respectively.

The operations are defined formally as follows.

Operation: PermitAccess
  Performed By: Reference Monitor
  Domain Pre-Condition:
    $\neg$ RM.permitaccess(s, o, r)
  Domain Post-Condition:
    RM.permitaccess(s, o, r)
  Input: subject, object, right
  ReqPre for [PermitToAccess]:
    PV.validate($p_1 \wedge \ldots \wedge p_n$)

Operation: PredicateValidation
  Performed By: Predicate Validator
  Domain Pre-Condition:
    $\neg$ PV.validate($p_1 \wedge \ldots \wedge p_n$)
  Domain Post-Condition:
    PV.validate($p_1 \wedge \ldots \wedge p_n$)
  Input: Predicate
  Output: ValidationResponse
  ReqPre for [PredicatesValidation]: tryaccess(s, o, r)

These operations can be considered as the *enforcement mechanism* for the UCON PreA$_0$ model of security policies. Indeed, the approach itself is to derive the specification of such enforcement mechanisms for any policy that the requirements specify using the KAOS operationalisation (patterns) presented in [24]. The semantics of the KAOS operations defines a set of proof obligations that lead to the realisation of the required *trigger*, *pre-* and *post-* conditions of a goal, and therefore the satisfaction of the goal itself. In this sense, and in our context, the proof of the semantics of an operation, in relation to the required conditions of a goal, validates the fact that the enforcement mechanism represented by the operation indeed implements (i.e. enforces) the corresponding UCON security policy expressed by the goal or requirement.

The only difference between these operations and those shown in Section 5 is in the specification of the *Required Pre-Condition* clause. This clause is required to ensure that the goals assigned to the individual agents are met. They are dependent on the order of the operations as specified by the model definition. Other UCON models encode a different sequentiality of the operations, and therefore may result in different required conditions. The rest of the operations definitions are the same as specified in Section 5. This *strategy-based* approach, is similar to that introduced in [22]. A possibility for the encoding of such strategy directly in the policy is the use of an operational policy language, such as for example a process algebra as that defined in [5, 10] with some notion of sequentiality. Alternatively, when writing UCON policies using other policy languages, another possibility would be to encode the strategy using an external scheduler, which is capable of implementing the right temporal ordering for the execution of the enforcement mechanism's operations.

6.2 Second Example: Refinement and Operationalisation of the UCON OnA$_3$ Policy

In the UCON OnA$_3$ core model, a usage control decision is determined by authorisations during the usage, and there is one or more attribute updates after this usage. We require the policy to be enforced before as well as throughout and after the access is permitted. The top goal is then the following, where we have used bounded temporal operators with the time range $[t_1, t_2]$ representing the period access takes place:

Goal [PermitOnA3]
RefinedTo: [PermitOnA3-pre], [PermitOnA3-on],
  [PermitOnA3-post]
FormalDef: ($\forall$ s:subject, o:object, r:right)

$$(\text{permit}\bar{\text{a}}\text{ccess(s,o,r)} \Rightarrow \blacklozenge_{<t_1}\text{policyEnforcing(s,o,r))} \wedge$$
$$(\Box_{t_1,t_2}\text{policyEnforcing(s,o,r))} \wedge (\Diamond_{>t_2}\text{policyEnforcing(s,o,r))}$$

We have assumed that t_1 is the time point at which access is granted, and t_2 is the point at which access ends. The first part of the conjunction states that the policy must be enforced prior to when access starts, i.e. at time t_1. The second part, on the other hand, states that the policy must be enforced all the time during the access, which is between the times t_1 and t_2. Finally, the third part states that the policy also must eventually be enforced after the access has ended, i.e. after time t_2. Note that for the case when pre-authorisation is not an issue, the first part simply becomes permitaccess(s,o,r) $\Rightarrow$ **True**.

We now apply a first goal refinement as shown in Fig. 7a. The formal definitions of the three immediate sub-goals are given as follows:

Goal [PermitOnA3-pre]
RefinedTo:
[Permit], [TryToAccess]
FormalDef:
($\forall$ s:subject, o:object,
r:right)
permitaccess(s, o, r) $\Rightarrow$
$\blacklozenge_{<t_1}$policyEnforcing(s, o, r)

Goal [PermitOnA3-on]
RefinedTo:
[CheckPredicates]
[ContinuousCheck]
FormalDef:
($\forall$ s:subject, o:object,
r:right)
permitaccess(s, o, r) $\Rightarrow$
$\Box_{t_1,t_2}$policyEnforcing(s, o, r)

Goal [PermitOnA3-post]
RefinedTo:
[EndOrRevoke], [Update]
[PostUpdateCompleted]
FormalDef:
($\forall$ s:subject, o:object,
r:right)
permitaccess(s, o, r) $\Rightarrow$
$\Diamond_{>t_2}$policyEnforcing(s, o, r)

These sub-goals split the main goal by cases; before, during and after enforcement of the UCON policy. The next level of refinement details more the enforcement of the policy in each case:

Goal [Permit]
Refines: [PermitOnA3-pre]
RefinedTo: [Permit Monitor/Control],
[PermitToAccess]
FormalDef: ($\forall$ s:subject, o:object, r:right)
permitaccess(s, o, r) $\Rightarrow$ $\bullet$tryaccess(s, o, r)

Goal [TryToAccess]
Refines: [PermitOnA3-pre]
FormalDef: ($\forall$ s:subject, o:object, r:right)
tryaccess(s, o, r) $\Rightarrow$ $\bullet$policyEnforcing(s, o, r)

Goal [CheckPredicates]
Refines: [PermitOnA3-on]
RefinedTo: [CP Monitor/Control],
[OnValidation]
FormalDef: ($\forall$ s:subject, o:object, r:right)
permitaccess(s, o, r) $\Rightarrow$ (p$_1$ $\wedge \ldots \wedge$ p$_n$)
$\mathcal{U}$ (revokeaccess(s, o, r) $\vee$ endaccess(s, o, r))

Goal [ContinuosCheck]
Refines: [PermitOnA3-on]
FormalDef: ($\forall$ s:subject, o:object, r:right)
(p$_1$ $\wedge \ldots \wedge$ p$_n$) $\mathcal{U}$ (revokeaccess(s, o, r) $\vee$
endaccess(s, o, r)) $\Rightarrow$ $\Box_{t_1,t_2}$policyEnforcing(s, o, r)

Goal [EndOrRevoke]
Refines: [PermitOnA3-post]
FormalDef:
($\forall$ s:subject, o:object,
r:right)
permitaccess(s, o, r) $\Rightarrow$
$\Diamond\big($ endaccess(s, o, r)
$\vee$ revokeaccess(s, o, r) $\big)$

Goal [Update]
Refines: [PermitOnA3-post]
RefinedTo:
[Update Monitor/Control],
[UpdateAfterEnd]
FormalDef:
($\forall$ s:subject, o:object,
r:right)
$\big($ endaccess(s, o, r)
$\vee$ revokeaccess(s, o, r) $\big)$ $\Rightarrow$
$\Diamond$update(s, o, r)

Goal [PostUpdateCompleted]
Refines: [PermitOnA3-post]
FormalDef:
($\forall$ s:subject, o:object,
r:right)
update(s, o, r) $\Rightarrow$
$\Diamond_{>t_2}$policyEnforcing(s, o, r)

The [Permit] and [TryToAccess] refinements of the pre-authorisation stage represent another sub-goal and an assumption on the domain, respectively. [Permit] states that to be permitted access to the resource, the subject must have tried in the previous state to access the resource with some right. [TryToAccess], on the other hand, states that trying to access a resource implies that some relevant (UCON) policy is in place to be applied. Note that since the policy is an on-going authorisation policy, there is no specific mention here of the kind of predicates needed to validate the pre-authorisation.

The next two sub-goals represent another domain assumption, [ContinuousCheck] and another sub-goal, [CheckPredicates]. [CheckPredicates] represents the on-going checks on the validity of the predicates, whereby such predicates must remain valid *until* access is revoked by the system or ended by the subject. This is a specific definition of on-going authorisation as was suggested by [8]. The righthand side of the implication represents a milestone for the refinement [24], and as such, it reappears in the domain assumption [ContinuosCheck], which states that the satisfaction of this milestone condition leads to the assumption that the UCON policy is being upheld throughout the whole period, t_1 to t_2, when access is taking place.

Finally, the last three sub-goals are the refinement of the post-authorisation stage, which include two domain assumptions [EndOrRevoke] and [PostUpdateCompleted], and another sub-goal [Update]. These three together again implement a milestone refinement strategy of their parent sub-goal. The first sub-goal, [EndOrRevoke], which is a domain assumption, states that there is an assumption that permitting an access will eventually lead to that access ending or being revoked. This then becomes the milestone for the next sub-goal, [Update], which states that such an end or revocation of the right to access is eventually followed by some update of the subject or resource's attributes. Finally, the domain assumption [PostUpdateCompleted] states that such an update of attributes assumes that the UCON policy will have been eventually enforced, beyond the completion time t_2 of the access.

The second part of Fig. 7b, shows how the refinement process is completed for the remaining sub-goals. This results in three more refinements, as defined below:

Goal [Permit Monitor/Control]
Refines: [Permit]
FormalDef: ($\forall$ s:subject,
o:object, r:right,
RM:Reference Monitor)
permitaccess(s, o, r) $\Leftrightarrow$
RM.permitaccess(s, o, r)

Goal [CP Monitor/Control]
Refines: [CheckPredicates]
FormalDef: ($\forall$ s:subject,
o:object, r:right,
PV:Predicate Validator)
($p_1 \wedge \ldots \wedge p_n$)
$\Leftrightarrow$ PV.validate
($p_1 \wedge \ldots \wedge p_n$)

Goal [Update Monitor/Control]
Refines: [Update]
FormalDef: ($\forall$ s:subject,
o:object, r:right,
AM:Attribute Manager)
update(s, o, r) $\Leftrightarrow$
AM.update(s, o, r)

Goal [PermitToAccess]
Refines: [Permit]
FormalDef: ($\forall$ s:subject,
o:object, r:right,
RM:Reference Monitor)
RM.permitaccess(s, o, r) $\Rightarrow$
$\bullet$tryaccess(s, o, r)
Resp: Reference Monitor

Goal [OnValidation]
Refines: [CheckPredicates]
FormalDef: ($\forall$ s:subject,
o:object, r:right,
PV:Predicate Validator)
permitaccess(s, o, r) $\Rightarrow$
PV.validate($p_1 \wedge \ldots \wedge p_n$) $\mathcal{U}$
(revokeaccess(s, o, r) $\vee$
endaccess(s, o, r))
Resp: Predicate Validator

Goal [UpdateAfterEnd]
Refines: [Update]
FormalDef: ($\forall$ s:subject,
o:object, r:right,
AM:Attribute Manager,
RM:Reference Monitor)
(endaccess(s, o, r) $\vee$
revokeaccess(s, o, r)) $\Rightarrow$
$\Diamond$ AM.update(s, o, r)
Resp: Attribute Manager

[Permit] is refined to [Permit Monitor/Control], which is another sub-goal, and [PermitToAccess], which is a requirement. [Permit Monitor/Control] states that permitting an access is dependant on the decision of the Reference Monitor (RM) component. This decision then itself depends on receiving, in a previous state, a request to access as represented by the tryaccess signal, in the requirement [PermitToAccess]. This latter requirement is assigned a responsible agent, called the Reference Monitor (RM). The next two sub-goals, again represent a sub-goal [CP Monitor/Control] and a requirement [OnValidation], both of which refine the parent sub-goal [CheckPredicates]. CP Monitor/Control states that the predicates required for the policy are always checked by a component called the Predicate Validator (PV). This then results in the requirement [OnValidation] being defined in terms of PV (Fig. 8).

Finally, the last pair of refined sub-goals represents a sub-goal, [Update Monitor/Control], which is a sub-goal refining [Update], and a requirement [UpdateAfterEnd], which is also refining [Update]. The first refined sub-goal, [Update Monitor/Control], states that the updating action is performed by a new component called the Attribute Manager (AM). This then allows us to redefine what an update is in terms of this component, leading to the definition of the [UpdateAfterEnd] requirement.

We are now capable of deriving the KAOS agent and operation models, that will operationalise the above three requirements. Figure 6.2 shows the KAOS agent/responsibility model, and the formal operational specification for the UCON OnA_3 enforcement mechanism.

As shown in the figure, there are three operations. These are `PermitAccess`, `PredicateValidation` and `AttributeUpdate`, derived using the KAOS operationalisation patterns, most of them presented in [24]. It should be noted that the formal specification of goals and operations allows the completeness, consistency and minimality of operationalisation to be formally verified. The semantics of the KAOS operations defines a set of proof obligations verifying that realising an operation when the required *trigger*, *pre-* and *post-* conditions of a goal are true implies the goal. In this sense, a proof of the semantics of each operation in relation to the required conditions validates that operations implement (i.e. enforce) the corresponding policies. The only difference between these operations and those shown in Section 5 is in the specification of the *Required Pre-Condition* and *Required Post-Condition* clauses. These clauses are required to ensure that the assigned requirements are met.

7 Related Work

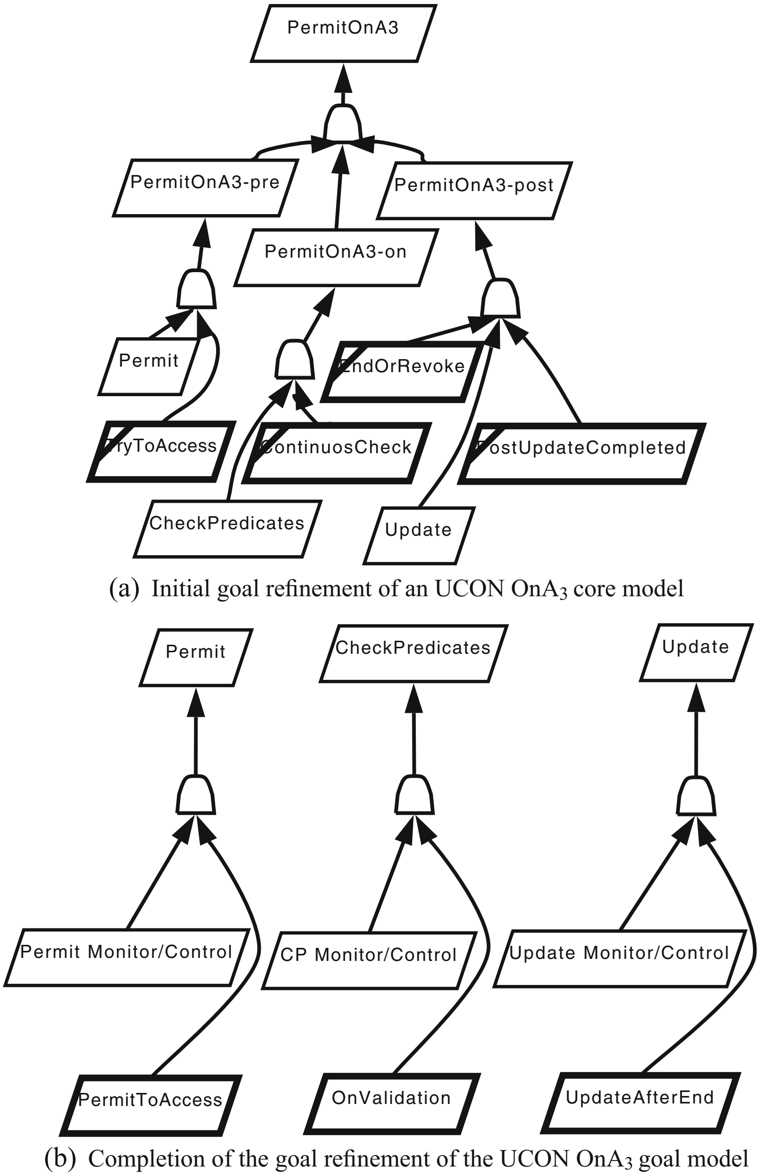
There are many approaches in the past (e.g. [25–28]) that have been proposed to control access and usage in a Grid environment, not all of which would be directly related to our approach. However, we mention here a few in order to contrast our work and approach with the aforementioned approaches.

In [25], the authors propose a platform (PRIMA) for the fine-grained management of access requests to Grid resources. While the approach is flexible in that access rights can be managed externally to a resource in a fine-grained manner, it does not consider the time continuity of such access requests, which is where our UCON-based approach fits in. In [26], usage control in Grids is considered at a higher level of abstraction; namely at the level of service level agreements. While this is complementary to our approach, it certainly does not provide the operational mechanisms (i.e.

the architecture) necessary for enforcing UCON-like policies. Other non-UCON access control approaches have also been proposed, including attribute-based access control (e.g. [27]) and role-based access control (e.g. [28]), however, these can be expressed in the UCON model as was outlines in [3, 8].

More realistically, this paper is associated with two strands of related work: policy refinement and derivation of enforcement mechanisms. The use of goal-refinement for refining policies as used here was introduced by Bandara et al. in [22]. The emphasis of [22] was on applying abduction techniques in order to determine the sequence of events needed to achieve a goal given a system architecture that already includes enforcement components. Close to Bandara's work is Loyola's work [29], in which also policies refinement is defined by applying requirement engineering and model checking techniques based on a temporal logic formalisation similar to the one used in this paper. The approach adopted in [22] allows one to find system executions aimed at fulfilling low-level goals that logically entail high-level strategic guidelines. From system executions, policy information is abstracted and eventually encoded into a set of refined policies

Fig. 7 Goal refinement of the UCON OnA$_3$ core model



(a) Initial goal refinement of an UCON OnA$_3$ core model

(b) Completion of the goal refinement of the UCON OnA$_3$ goal model

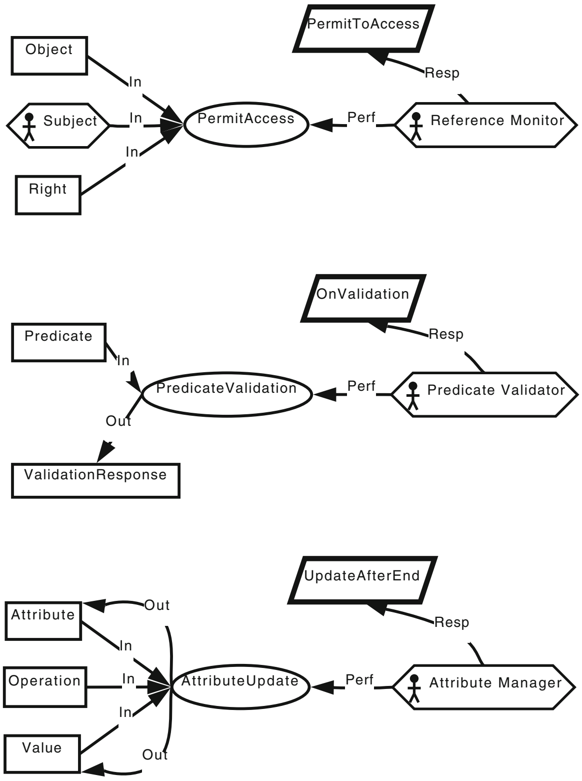


Fig. 8 KAOS agent and operation models for the enforcement mechanism for UCON OnA_3

specified in Ponder. All the above approaches have been applied to the networking management domain.

An alternative approach is presented by Chadwick et al. in [30], based on the existence of a resource hierarchy. Their work exploits Semantic-Web technology to automate the refinement process. We consider the representation of a resource hierarchy as an interesting idea and plan to study as future work the inclusion of resource hierarchy in goal-based approaches to policy refinement.

In relation to the derivation of enforcement mechanisms, Janicke et al. present in [31] a framework for the derivation of enforcement mechanisms that guarantees compliance with the policies. Their work is based on formalising the policies in Interval Temporal Logic (ITL) and concentrates only on history-based access control policies. Our work is more operational and we consider that our work can be linked better to existing efforts in implementing usage control for Grids that we reviewed in Section 3. Nonetheless, our approach suffers from the lack of a direct method for expressing the sequential order in which the derived operations related to a particular UCON policy will need to be executed. Despite the fact that such encoding is possible (as a additional layer of expressivity), the approach would benefit from enhancing the operational model with a high-level language for expressing such *execution strategies*. As such, more low-level operational languages, such as POLPA [5], adopt a different approach whereby the policy specification itself encodes the execution strategy (i.e. the temporal order in which operations are executed.) Another possibility in our case would be to use an external scheduler, conscious of the UCON sub-model the policy to be enforced pertains to.

8 Conclusion and Future Work

This paper presented a fine-grained access and usage control Grid authorisation architecture with strong reference to the OGSA work [17], and a rigorous approach to the design of an enforcement mechanism for usage control policies. We concentrated on the UCON_{abc} model proposed by Park and Sandhu [3] and studied its application for the case of Grids. Our approach consists in applying the KAOS requirements engineering methodology to the derivation of the enforcement mechanism, based on the treatment of UCON policies as requirements. The UCON policies can be refined into concrete requirements (that could be enforced by the resulting system) by repeatedly applying the KAOS goal refinement process. KAOS offers a formal language to represent a goal/requirement based on temporal logic, which is similar to the formal language used to give semantics to UCON models [3, 8]. The refinement method also includes strategies and patterns to guide the refinement process, and there is tool support aiding the user in this process [23]. We assigned the low-level policies to software agents responsible for executing the operations that enforce the concrete policies. Our resulting architecture consists of three agents: an *Attribute Manager*, responsible for updating attributes associated to subjects and objects, a *Predicate Validator*, responsible for validating policy predicates and a *Reference Monitor*, acting as a gateway for all the usage decisions in the Grid.

In future works, we plan to use KAOS to formally analyse the requirements for a general enforcement mechanism that covers the complete range of the

UCON$_{abc}$ policies, thus including not only authorisations (i.e. the UCON$_a$ family of models), but also obligations (the UCON$_b$ model family) and conditions (the UCON$_c$ model family). We plan to work on the definition of an architecture with the final objective of either propose a prototype, or extend the already developed implementations for Grid authorisation systems (e.g. the OGF architecture). We shall also review the already deployed policy languages and analyse their capacity to encode UCON policies, keeping in consideration the OGF recommendation on the use of standards. Finally, we plan to extend this work to the domain of Cloud computing by proposing an architecture and a model for controlling access and usage for Cloud infrastructures.

References

1. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: enabling scalable virtual organizations. International Journal of Supercomputer Applications **15**(3) (2001)
2. Venugopal, S., Buyya, R., Ramamohanarao, K.: A taxonomy of data grids for distributed data sharing, management, and processing. ACM Comput. Surv. **38**(1), 3 (2006)
3. Park, J., Sandhu, R.: The UCON$_{ABC}$ usage control model. ACM Trans. Inf. Syst. Secur. **7**(1), 128 (2004)
4. Pretschner, A., Hilty, M., Basin, D.: Distributed usage control. Commun. ACM **49**(9), 39 (2006)
5. Martinelli, F., Mori, P.: A model for usage control in GRID systems. In: Grid-STP2007, International Conference on Security, Trust and Privacy in Grid Systems. IEEE Computer Society (2007)
6. Zhang, X., Nakae, M., Covington, M.J., Sandhu, R.: Toward a usage-based security framework for collaborative computing systems. ACM Trans. Inf. Syst. Secur. **11**(1), 3:1 (2008)
7. van Lamsweerde, A.: Requirements Engineering in the Year 00: A Research Perspective. In: International Conference on Software Engineering, pp. 5–19 (2000)
8. Zhang, X., Parisi-Presicce, F., Sandhu, R., Park, J.: Formal model and policy specification of usage control. ACM Trans. Inf. Syst. Secur. **8**(4), 351 (2005)
9. Sandhu, R., Park, J.: Usage control: A vision for next generation access control. In: MMM-ACNS, pp. 17–31 (2003)
10. Martinelli, F., Mori, P.: On usage control for GRID systems. Futur. Gener. Comput. Syst. **26**(7), 1032 (2010)
11. Naqvi, S., Massonet, P., Aziz, B., Arenas, A., Martinelli, F., Mori, P., Blasi, L., Cortese, G.: Fine-grained continuous usage control of service based grids - The GridTrust approach. In: Proceedings of the 1st European Conference on Towards a Service-Based Internet, Springer-Verlag, ServiceWave'08, pp. 242–253 (2008)
12. OASIS: Oasis Extensible Access Control Markup Language (XACML), http://www.oasis-open.org/committees/xacml (2005)
13. e Ghazia, U., Masood, R., Shibli, M.A., Bilal, M.: Usage control model specification in XACML policy language. In: Proceedings of the 11th IFIP TC 8 International Conference on Computer Information Systems and Industrial Management, Springer-Verlag, CISIM'12, pp. 68–79 (2012)
14. Colombo, M., Lazouski, A., Martinelli, F., Mori, P.: A proposal on enhancing XACML with continuous usage control features. In: Desprez, F., Getov, V., Priol, T., Yahyapour, R. (eds.) Grids, P2P and Services Computing, pp. 133–146. Springer (2010)
15. Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E., Martinelli, F., Mori, P.: Testing of PolPA-based usage control systems. Softw. Qual. Control **22**(2), 241 (2014)
16. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: Enabling scalable virtual organizations. Int. J. High Perform. Comput. Appl. **15**(3), 200 (2001)
17. Chadwick, D.: Functional Components of Grid Service Provider Authorisation Service Middleware. Technical Report, Open Grid Forum (2008)
18. van Lamsweerde, A.: Requirements engineering - from system goals to UML models to software specifications. Wiley (2009)
19. Vardi, M.Y.: Branching vs. linear time: Final showdown. In: Margaria, T., Yi, W. (eds.) Proceedings of the 7th International Conference On Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001), Lecture Notes in Computer Science, vol. 2031, pp. 1–22. Springer (2001)
20. Objectover: A Power Tool to Engineer Your Business and Technical Requirements. http://www.objectiver.com/fileadmin/download/documents/leaflet.pdf (2015)
21. Moffett, J., Sloman, M.: Policy Hierarchies for Distributed Systems Management. IEEE J. Selected Areas in Communications **11**(9), 14 04 (1993)
22. Bandara, A.K., Lupu, E.C., Moffett, J., Russo, A.: A goal-based approach to policy refinement. In: 5th IEEE Workshop on Policies for Distributed Systems and Networks. IEEE Computer Society (2004)
23. Ponsard, C., Massonet, P., Molderez, J.F., Rifaut, A., van Lamsweerde, A., Hung, T.V.: Early verification and validation of mission critical systems. J. Form. Methods Syst. Des. **30**(3) (2007)
24. Letier, E., van Lamsweerde, A.: Deriving operational software specifications from system goals. In: FSE'10: 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (2002)
25. Lorch, M., Kafura, D.: The PRIMA grid authorization system. J. Grid Comput. **2**(3), 279 (2004)
26. Dumitrescu, C.L., Raicu, I., Foster, I.: The design, usage, and performance of GRUBER: A grid usage service level

agreement based brokERing infrastructure. J. Grid Comput. **5**(1), 99 (2007)

27. Lang, B., Foster, I., Siebenlist, F., Ananthakrishnan, R., Freeman, T.: A flexible attribute based access control method for grid computing. J. Grid Comput. **7**(2), 169 (2009)
28. Muppavarapu, V., Chung, S.: Role-based access control in a data grid using the storage resource broker and shibboleth. J. Grid Comput. **7**(2), 265 (2009)
29. Rubio-Loyola, J., Serrat, J., Charalambides, M., Flegkas, P., Pavlou, G., Lafuente, A.: Using linear temporal model checking for goal-oriented policy refinement frameworks. In: 6th IEEE International Workshop on Policies for Distributed Systems and Networks, pp. 181–190 (2005)
30. Su, L., Chadwick, D., Basden, A., Cunningham, J.: Automated decomposition of access control policies. In: 6th IEEE International Workshop on Policies for Distributed Systems and Networks, pp. 3–13. IEEE Computer Society (2005)
31. Janicke, H., Cau, A., Siewe, F., Zedan, H.: Deriving Enforcement Mechanisms from Policies. IEEE Computer Society (2007)